Taming Large Models with Hawk and NeoEMF

A. García-Domínguez, D. S. Kolovos, K. Barmpis, G. Daniel, G. Sunyé

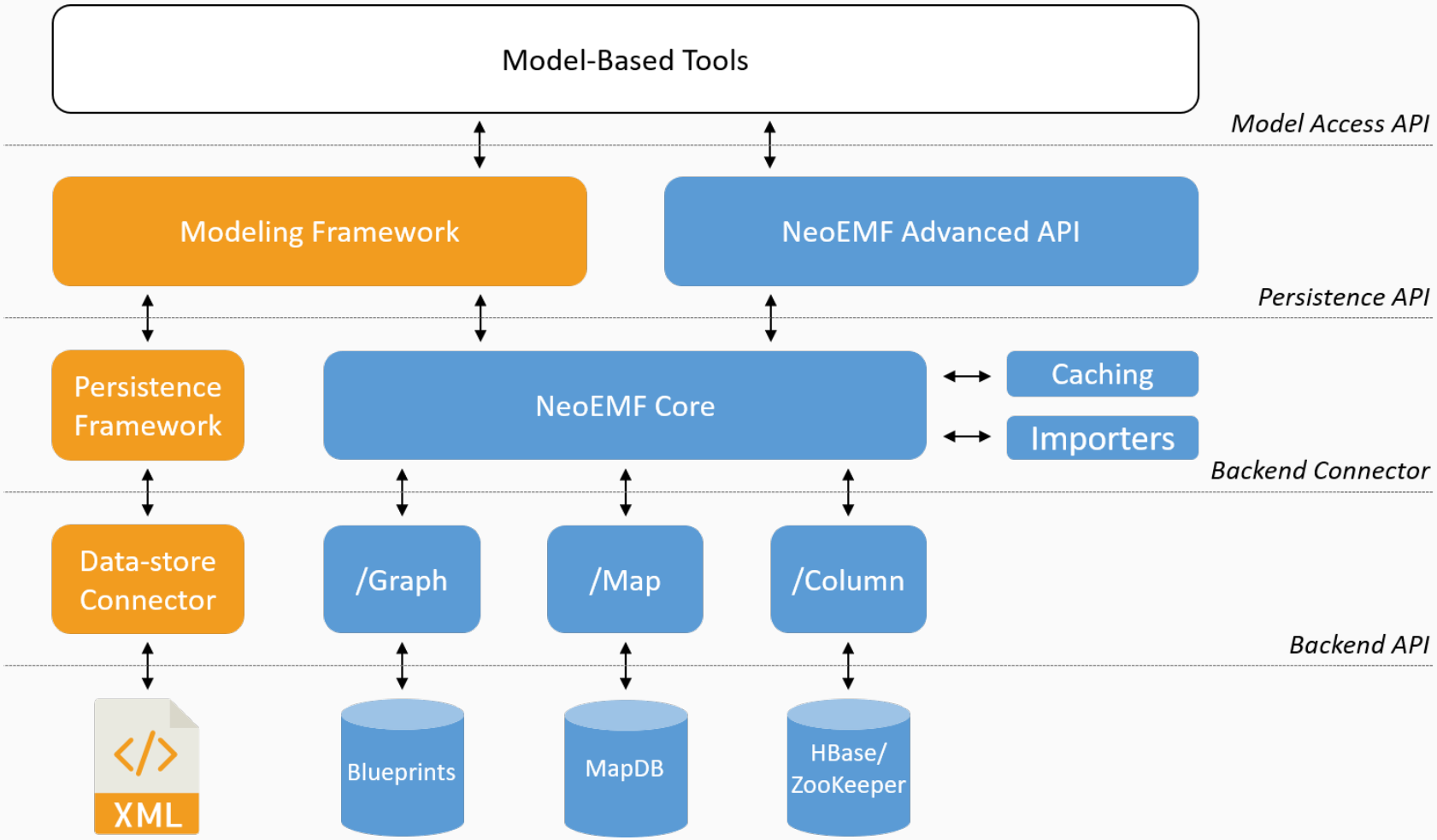
MoDELS'2018, 14–19 October 2018

NeoEMF

NeoEMF: overview

- Handle large models with task-specific databases
- Lazy-loading
- Compliant with the EMF API
  - Easy to integrate in existing applications
  - EMF-Compatible code generation
- Advanced caching (& prefetching strategies)
- Efficient XMI importer

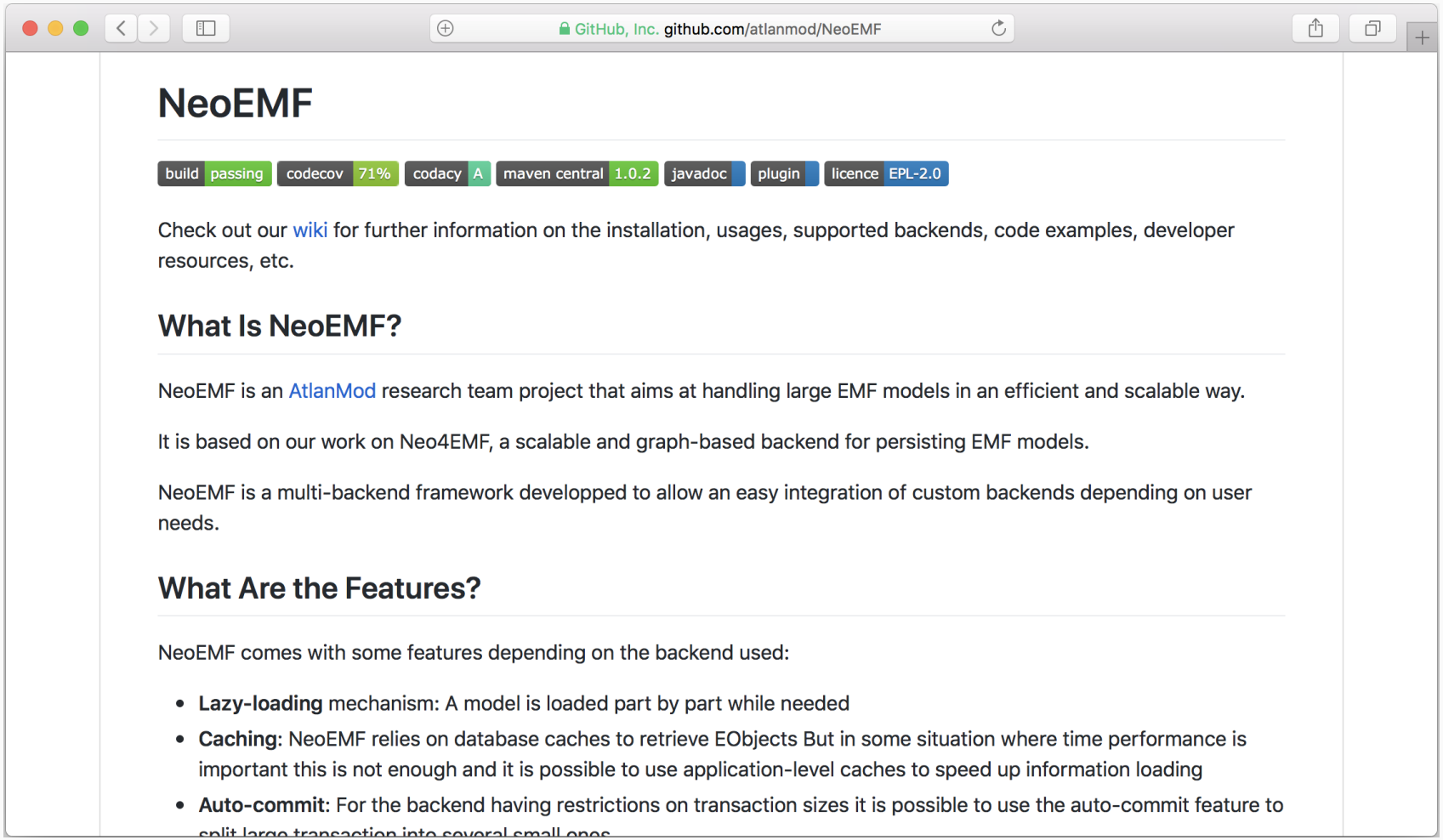
NeoEMF: Architecture



NeoEMF: datastores [DSB+17]

- **NeoEMF/Graph**
  - Efficient model traversal using rich query language
  - Mogwaï framework (OCL to Gremlin translation)
- **NeoEMF/Map**
  - Fast access to atomic operations
  - Designed for EMF-API calls
- **NeoEMF/Column**
  - Transparent model distribution
  - Concurrent read/write
  - Distributed model transformations (ATL-MR)

NeoEMF: project website



- `https://github.com/atlanmod/NeoEMF`
- Open source project under the Eclipse Public License 2.0

NeoEMF: initialise a new resource

1. Register the Persistence Backend Factory.
2. Create a ResourceSet and register the PersistentResourceFactory
3. Create a new URI to locate a file-based resource.
4. Create the resource.

```
PersistenceBackendFactoryRegistry.register(
        BlueprintsURI.SCHEME,
        BlueprintsPersistenceBackendFactory.getInstance());

ResourceSet resourceSet = new ResourceSetImpl();
resourceSet.getResourceFactoryRegistry().getProtocolToFactoryMap()
        .put(BlueprintsURI.SCHEME,
                PersistentResourceFactory.getInstance());

URI uri = BlueprintsURI.createFileURI(new File("<db_path>"));

Resource resource = resourceSet.createResource(uri);

// EMF resource stored in an in-memory Blueprints graph
```

NeoEMF: persist a resource

1. Create a new option builder (backend-specific).
2. Save the resource.
3. Manipulate the resource by accessing the local database

```
Map<String, Object> options = BlueprintsNeo4jOptionsBuilder.newBuilder()
        .weakCache().autocommit().cacheSizes()

resource.save(options);
// Resource saved in a local Neo4j database
resource.getContents()
// [...]
```

NeoEMF: modify an existing resource

1. Load resource using the same option builder.
2. Navigate its content and perform update operations
3. Save the resource (automatically done with *autocommit* option)

```
Map<String, Object> options = BlueprintsNeo4jOptionsBuilder.newBuilder()
        .weakCache().autocommit().cacheSizes()

URI uri = BlueprintsURI.createFileURI(new File("<db_path>"));

resourceSet.createResource(uri)
resource.load(options);

// Model manipulation operation (complete EMF API support)
MyClass myClass = (MyClass) resource.getContents().get(0);
myClass.setName("NewName");

// Save the modification in the local Neo4j database
resource.save(config.asMap());
```

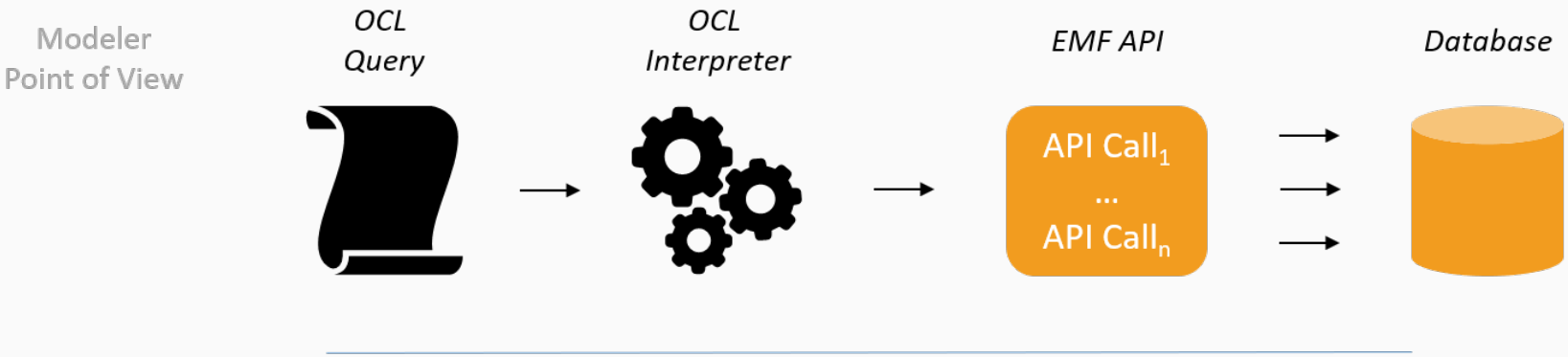
Demo time!

Let's import a Java model, save it in Neo4j and MapDB, and query the database.

Mogwaï

Mogwaï: motivation

- NeoEMF improves model scalability, but ...



Under the Hood

Package.allInstances().name → get(p1) get(p1,name) ... get(pn) get(pn,name)

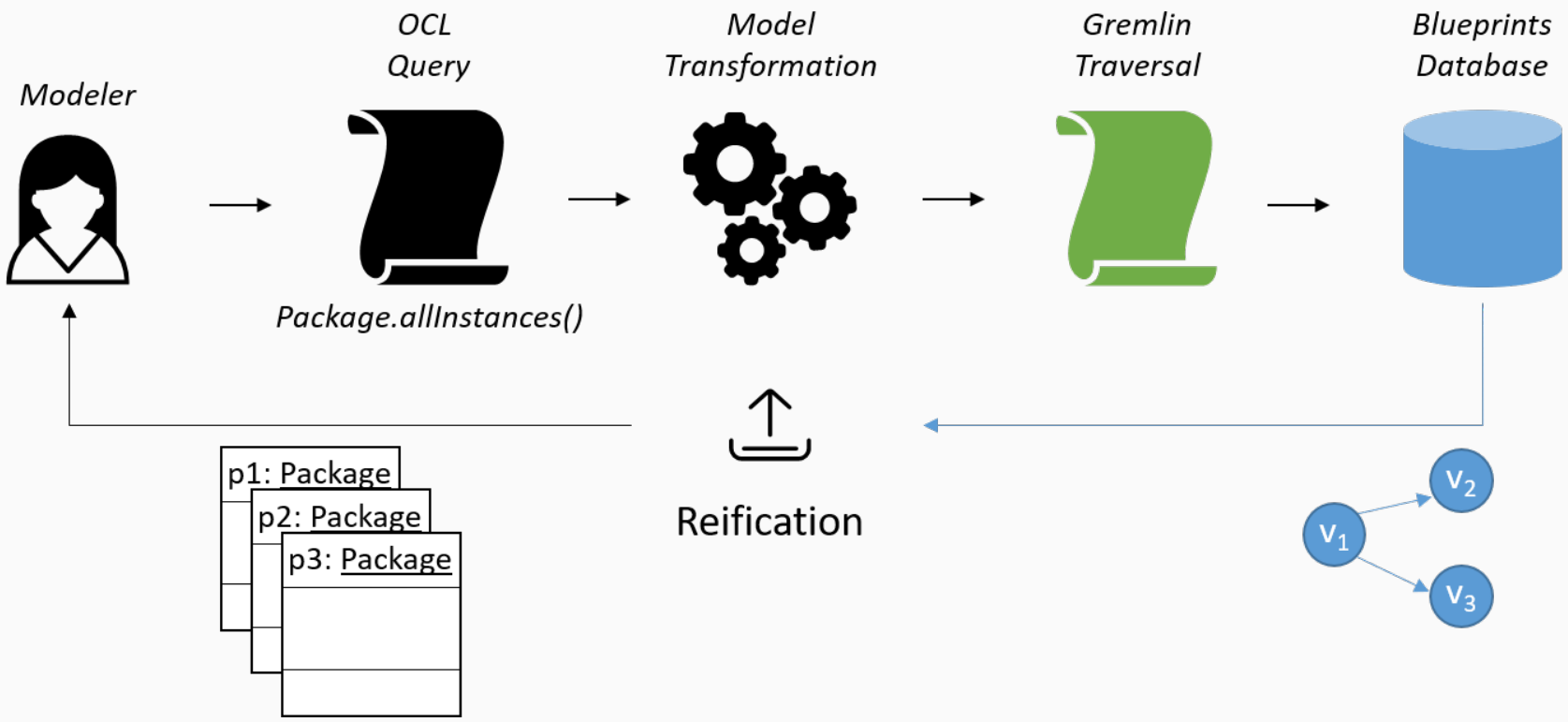
Mogwaï: motivation

- Low-level model handling APIs
  - Not aligned with the database capabilities
- Fragmented queries on the database
  - Not efficient
  - Remote databases
- Intermediate object reification
  - Memory consumption
  - Execution time overhead

Mogwaï: motivation

- Database queries are efficient but
  - Modern persistence frameworks typically rely on NoSQL databases
    - Multiple query languages
    - Multiple data representations
    - Low-level queries are hard to understand and maintain
    - Modeling expertise vs. Database expertise
- Solution: generate them!

Mogwaï: architecture [DSC16]



- Generate graph database queries from OCL expressions
- Bypass modelling framework API
- Single execution of the query
- "Compatible" with EMF

Mogwaï: under the hoods

- Gremlin metamodel (around 100 classes)
- ATL Transformation
  - OCL-to-Gremlin mapping
  - Query composition
  - 70 rules and helpers
- Customized Gremlin engine
- Model element reification mechanism

Mogwaï: load a Mogwaï resource

1. Register the BlueprintsPersistenceBackendFactory.
2. Create a ResourceSet and register the PersistentResourceFactory
3. Create a new MogwaiURI to locate a file-based resource.
4. Create and cast the resource.
5. Use the Mogwaï API

```
PersistenceBackendFactoryRegistry.register(
    MogwaiURI.MOGWAI_SCHEME, BlueprintsPersistenceBackendFactory.getInstance());

ResourceSet resourceSet = new ResourceSetImpl();
resourceSet.getResourceFactoryRegistry().getProtocolToFactoryMap()
    .put(MogwaiURI.MOGWAI_SCHEME, MogwaiResourceFactory.getInstance());

URI uri = MogwaiURI.createMogwaiURI(new File("<db_path>"));

MogwaiResource resource = (MogwaiResource) resourceSet.createResource(uri);
resource.load(Collections.emptyMap());

// Use EMF Resource with enhanced querying API
resource.query([...]);
```

Mogwaï: load and execute an OCL query

1. Create a MogwaiQuery
2. Query the resource and get a QueryResult
3. Retrieve the database results, execution time, executed query ...

```
MogwaiQuery query = OCLQueryBuilder.newBuilder()
    .fromURI(URI.createURI("ocl/singletonMethods.ocl"))
    .build();

QueryResult result = resource.query(MogwaiQuery);
result.isSingleResult(); // returns only one element?
result.resultSize(); // number of results
result.getExecutedQuery(); // get the executed database query
result.getComputationTime(); // time to compute the query
result.getResults(); // Collection<Object> of database results
```

Mogwaï: manipulate query results

1. Get a NeoEMFQueryResult
2. Reify the results (if possible[1])
3. Navigate your model elements using the standard EMF API

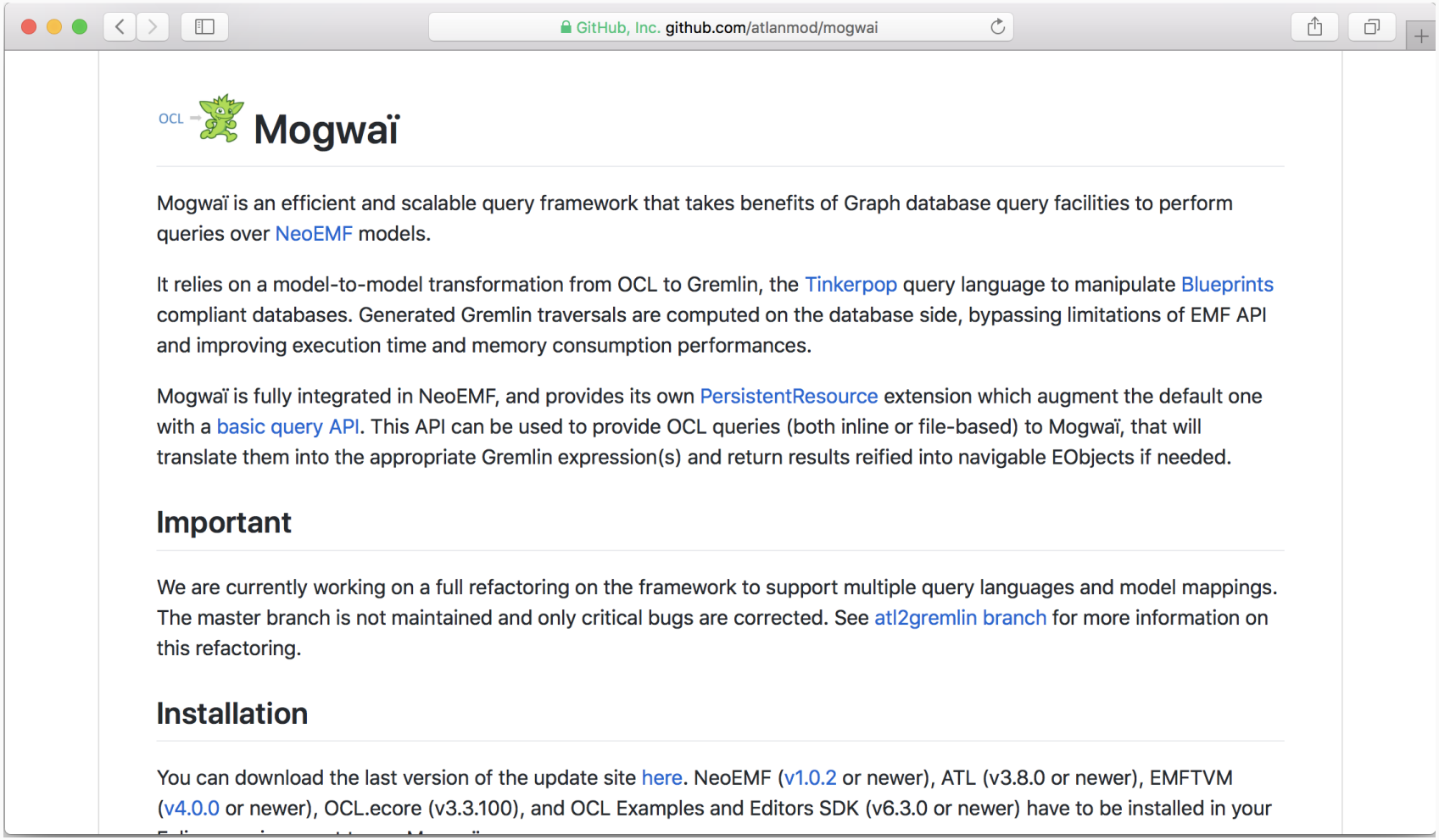
```
MogwaiQuery query = OCLQueryBuilder.newBuilder()
    .fromURI(URI.createURI("ocl/singletonMethods.ocl"))
    .build();

NeoEMFQueryResult result = resource.query(MogwaiQuery);
if(result.isReifiable()) {
    List<EObject> eObjects = result.reifyResults();
    for(EObject e : eObjects) {
        System.out.println(((MethodDeclaration)e).getName());
    }
}
```

[1] Primitive types cannot be reified

Mogwaï: new features

- ModelDatastore abstraction
  - Support for different data stores
  - Easily extensible
  - Generic queries
- Prototype support for model transformations (Gremlin-ATL [DJSC17])
- Data migration operations
- Large model validation (presenting this work at 15:00 at OCL'18)

Mogwaï: project website



- `https://github.com/atlanmod/mogwai`
- Open source project under the Eclipse Public License 2.0

NeoEMF & Mogwaï: summing up

NeoEMF

- Select the NoSQL database adapted to a modeling scenario
- Transparent EMF integration
- On-demand loading

Mogwaï

- Benefit from the capabilities of NeoEMF/Graph backend
- Translates OCL queries into Gremlin traversals
- Bypasses low-level modeling APIs

References i

Gwendal Daniel, Frédéric Jouault, Gerson Sunyé, and Jordi Cabot. **Gremlin-ATL: a scalable model transformation framework.** In *Automated Software Engineering (ASE), 2017 32nd IEEE/ACM International Conference on*, pages 462–472. IEEE, 2017.

Gwendal Daniel, Gerson Sunyé, Amine Benelallam, Massimo Tisi, Yoann Vernageau, Abel Gómez, and Jordi Cabot. **NeoEMF: a multi-database model persistence framework for very large models.** In *Science of Computer Programming*, pages 1–7. Elsevier, 2017.

References ii


Gwendal Daniel, Gerson Sunyé, and Jordi Cabot.
Mogwaï: A framework to handle complex queries on large models.
In *Tenth IEEE International Conference on Research Challenges in Information Science, RCIS 2016, Grenoble, France, June 1-3, 2016*, pages 1–12. IEEE, 2016.